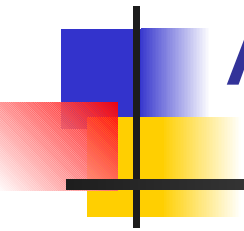


Aliasing in GCC

Daniel J Berlin

Quick intro to GCC IR's

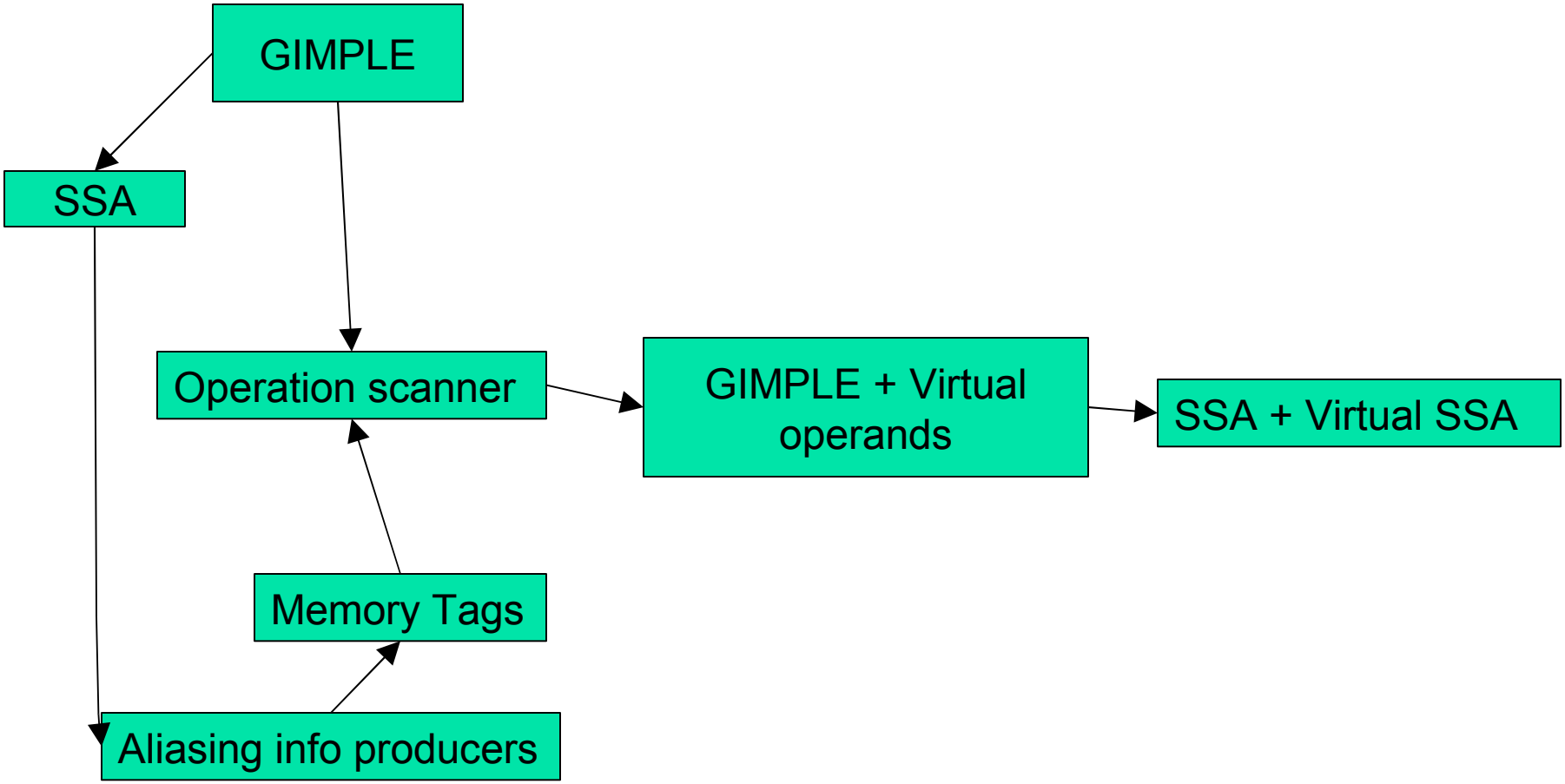
- GIMPLE - High level IR
  - Three address code
  - Simplified operations
  - Easy to analyze
  - Came second (GCC 4.0)
- RTL - Low level IR
  - LISP like
  - Complex operations
  - Hard to analyze
  - Came first (GCC 1.0)

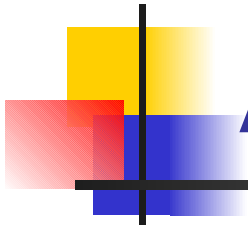
```
int main (void)
{
  int D.1517,D.1518,D.1519;
  int a,b,c;

  D.1517 = b + c;
  D.1518 = D.1517 + c;
  a = D.1518 + c;
  D.1519 = a;
  return D.1519;
}
```

```
(insn 10 (set (reg:SI 123)
  (plus:SI (reg/v:SI 120 [ b ])
  (reg/v:SI 119 [ c ]))))
(insn 11 (set (reg:SI 124)
  (plus:SI (reg:SI 123)
  (reg/v:SI 119 [ c ]))))
(insn 12 (set (reg:SI 122)
  (plus:SI (reg:SI 124)
  (reg/v:SI 119 [ c ]))))
(insn 13 (set (reg:SI 121 [ <result> ])
  (reg:SI 122)))
```

Overview of Tree Level Aliasing





Aliasing Info Producers

- Structure alias info producer (4.1)
  - Disambiguate structure fields
  - Disambiguate small arrays (4.2)
  - One tag for each distinct field in a structure
  - Very precise

Structure alias info producer

```
struct foo
{
  int a;
  int b;
  int c;
};

struct foo bar;
```

Would normally have one memory tag:

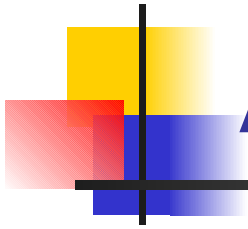
SMT.0 aliases { bar }

Instead we will produce one for each field:

SFT.0 aliases {bar.a}

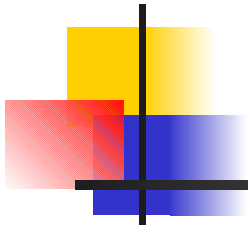
SFT.1 aliases {bar.b}

SFT.2 aliases {bar.c}



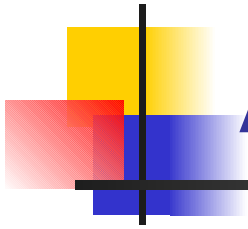
Aliasing Info Producers

- Points-to analyzer
  - Field-sensitive intraprocedural algorithm (4.1)
  - Field-sensitive context-insensitive interprocedural algorithm (4.2)
  - Precise when it comes up with usable answers



Points-to analyzer

- Build weighted constraint graphs from source code and solve them
- Uses various speedup techniques
  - Variable substitution
  - Cycle elimination
  - Proper ordering



Aliasing Info Producers

- Type based alias analysis
  - Uses language rules to disambiguate
  - Everything falls back to this
  - Very imprecise

```
void foo(int *a, float *b)
{
  *a = 5;
  *b = 7.0f;
}
```

These two stores cannot alias in C

Memory Tags

- The heart of the tree level aliasing system
- Each tag has an associated set of aliases
- Each pointer may have multiple memory tags

```
int *foo(int a, int b)
{
  ptr1 = &b;
  if (a)
    ptr2 = &a;
  ptr3 = φ(ptr1, ptr2)
  return ptr3
}
```

Memory tag NMT.1 aliases {a, b}

Memory tag NMT.2 aliases {a}

Memory tag NMT.3 aliases {b}

Variable ptr has memory tag NMT.1

SSA variable ptr_1 has memory tag NMT.3

SSA variable ptr_2 has memory tag NMT.2

SSA variable ptr_3 has memory tag NMT.1

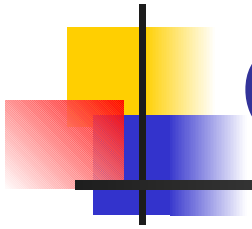
Operation Scanner

- Looks for loads and stores
- Prunes memory tag aliases (4.2)
- Generates GIMPLE with virtual operations using memory tags

```
int foo(int *p, int *q)
{
  *p = 5
  return *q;
}
```

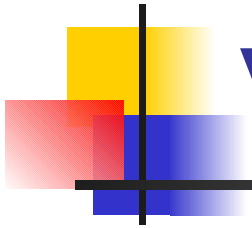
```
int foo(int *p, int *q)
{
  SMT.5 = V_MAY_DEF <SMT.5>
  *p = 5

  VUSE <SMT.5>
  return *q;
}
```



Call Clobbering

- Interprocedural pass for statics (4.1)
- Intraprocedural pass for everything else

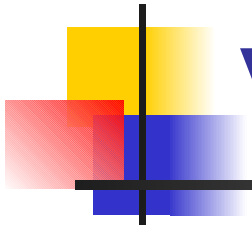


Virtual SSA

- SSA for memory and other operations that can't be directly renamed

```
struct foo { int a; int b; } bar;
int maybeclobbered;

int foo(struct foo *incoming)
{
        /* Does bar get renamed here?
           It depends. */
        incoming->a = 5
        /* Does maybeclobbered or bar need to be renamed here?
           It depends.  */
        globalkiller();
}
```



Virtual SSA

- Building SSA over virtual operands solves this in a conservatively correct way.

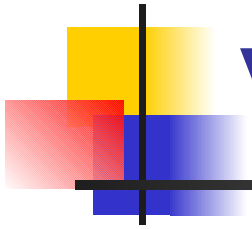
```
struct foo { int a; int b; } bar;
int maybeclobbered;

int foo(struct foo *incoming)
{
        SFT.1_1 = V_MAY_DEF <SFT.1_0>
        incoming->a = 5

        SFT.1_2 = V_MAY_DEF <SFT.1_1>
        SFT.2_1 = V_MAY_DEF <SFT.2_0>
        SMT.1_1 = V_MAY_DEF <SMT.1_0>
        globalkiller();
}
```

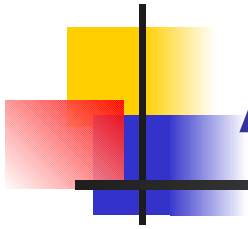
Memory tags:

SFT.1 aliases { bar.a }
SFT.2 aliases { bar.b }
SMT.1 aliases { maybeclobbered }



Virtual SSA

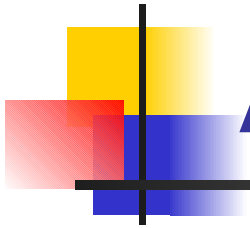
- Consumers use virtual SSA as a **base** for memory optimization
  - Code sinking
    - Stores can sink closer to uses
  - Value based Load PRE
    - Value numbering memory operations based on vuses associated with them
  - Other memory optimizations



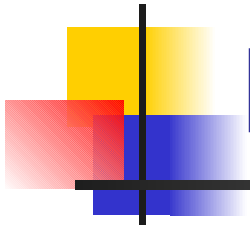
All is not a bed of roses

- Time usage
- Space usage
- Updating SSA
- Interprocedural info

Overview of RTL Level Aliasing

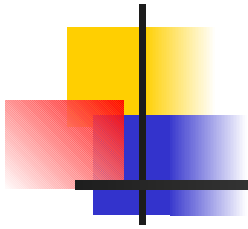


- Less of an actual designed system
- Somewhat convoluted
- Useful information is not preserved from tree level



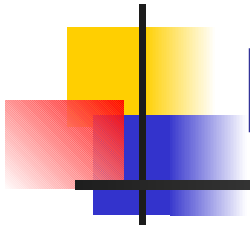
RTL Interface to aliasing

- No direct method of determining set of aliases
- Simple pairwise query system
  - Built out of disambiguators
  - Do stores/reads to X affect stores/reads to Y



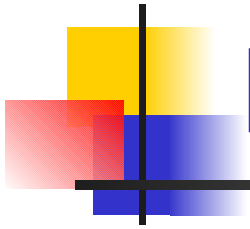
RTL aliasing - producing info

- RTL expansion attempts to associate some useful info with memory
  - (MEM (SYMBOL_REF “foo”) [2 argc+0 S4 A32])
  - (MEM (PLUS (REG) (4)) [3 <variable>.a.b.c S4 A8])
- RTL attempts to divine some information on its own
  - Tracking accessed memory address by propagating values through registers
  - Other information tracked to



RTL aliasing - usage

- Passes explicitly request alias analysis information
  - Each call recomputes all information RTL divines on its own
  - Usually okay about keeping info up to date
- Other passes destroy it with reckless abandon



RTL aliasing - problems

- Destruction by passes
- Lack of preserving tree level information
- Everything is always clobbered

- People are working on all of these problems